Outline

Binary Search Tree (BST): A Quick Revision

The Importance of a Balanced BST
• To keep $h = O(\log n)$

Adelson-Velskii Landis (AVL) Tree
• Principle of “Height-Balanced”
• Keeping AVL Tree balanced via rotations
• Code is shown but not given (try this during PS2)

Relation with CS2010 PS2: “The Baby Names Problem”

Reference in CP3 book: Page 43-47 + 380-382
Vertex x has two children: `x.left`, `x.right` and one parent: `x.parent`
  - x.left/x.right/x.parent can be null for some vertices

- Vertex x has a key: `x.key`

- **BST Property**: all keys in left sub-tree < `x.key` < all keys in right sub-tree
BST Web-based Review

http://visualgo.net/bst.html

(root)

(internal vertices)

(leaves)
More BST Attributes: Height and Size

Two more attributes at each BST vertex: Height and Size

- **Height**: \#edges on the path from this vertex to deepest leaf
- **Size**: \#vertices of the subtree rooted at this vertex

These values can be computed recursively:

- \( x\.height = -1 \) (if \( x \) is an empty tree)
- \( x\.height = \max(x\.left\.height, x\.right\.height) + 1 \) (all other cases)
- \( x\.size = 0 \) (if \( x \) is an empty tree)
- \( x\.size = x\.left\.size + x\.right\.size + 1 \) (all other cases)

The height of the BST is thus: \( \text{root\.height} \)

The size of the BST is thus: \( \text{root\.size} \)
Binary Search Trees: Height \( h \)
Binary Search Trees: Size (s)

- Root: 41
  - Left: 20 (s=4)
    - Left: 11 (s=1)
      - Left: None
      - Right: None
    - Right: 29 (s=2)
      - Left: None
      - Right: None
  - Right: 65 (s=5)
    - Left: None
    - Right: 50 (s=1)
      - Left: None
      - Right: 72 (s=1)
        - Left: None
        - Right: 91 (s=3)
          - Left: None
          - Right: None
    - Right: None
The height of this tree is?

1. 2
2. 4
3. 5
4. 6
5. 7
6. 42
The size of this tree is?

1. 10
2. 11
3. 12
4. 13
5. 14
6. 15
Binary Search Tree: Summary

Operations that **modify** the BST (*dynamic* data structure):
- insert: \( O(h) \)
- delete: \( O(h) \)

Query operations (the BST structure remains the same):
- search: \( O(h) \)
- findMin, findMax: \( O(h) \)
- predecessor, successor: \( O(h) \)
- inorder traversal: \( O(n) \) – the only one that does not depend on \( h \)
  - PS: We also have preorder and postorder traversals for tree structure (discussed in tutorial)
- select/rank: ? (we have not discuss this yet)
Most operations take $O(h)$ time
Lower bound: $h > \log_2(n)$

$n \leq 1 + 2 + 4 + \ldots + 2^h$

$\leq 2^0 + 2^1 + 2^2 + \ldots + 2^h < 2^{h+1}$ (sum of geometric progression)

$\log_2(n) < \log_2(2^{h+1}) \Rightarrow \log_2(n) < (h + 1) \times \log_2(2) \Rightarrow h > \log_2(n) - 1$

$\Rightarrow h > \log_2(n)$
Most operations take $O(h)$ time
Upper bound: \( h \leq n-1 \implies h < n \)
The Importance of Being Balanced

Most operations take $O(h)$ time

Combined bound: $\log_2(n) < h < n$

$log_2(n)$ versus $n$ in picture (revisited with larger numbers):

- $n = 500$
  - $\log_2(n) \sim 9$
  - If we just stop at CS1020

- $n = 1000$
  - $\log_2(n) \sim 10$
  - After learning CS2010 😊

We say a BST is balanced if $h = O(\log n)$, i.e. $O(c \times \log n)$

On a balanced BST, all operations run in $O(\log n)$ time
The Importance of Being Balanced

Example of a perfectly balanced BST:
This is hard to achieve though...
The Importance of Being Balanced

How to get a balanced tree:

– Define a good property of a tree
– Show that if the good property holds, then the tree is balanced
– After every insert/delete, make sure the good property still holds
  • If not, fix it!
Adelson-Velskii & Landis, 1962 (~53 years ago... :O)

Can be a little bit frustrating if you are not comfortable with recursion
Hang on...

**AVL TREES**
Step 1: Augment (i.e. add more information)

In every vertex $x$, we also store its height: $x$.height

(Note that $x$ already has: $x$.left, $x$.right, $x$.parent, and $x$.key)

During insertion and deletion, we also update height:

```cpp
insert(x, v)
    // ... same as before ...
    x.height = max(x.left.height, x.right.height) + 1
    // update height during deletion too (same as above)
```
Binary Search Trees

**insert(27)**

Height of empty trees are ignored in this illustration (all -1)

Height information during insertion/deletion is not shown in VisuAlgo (yet)
Binary Search Trees

insert(27)
Binary Search Trees

insert(27)

Notice that only vertices along the insertion path may have their height attribute updated...
AVL Trees [Adelson-Velskii & Landis 1962]

Step 2: Define Invariant (something that will not change)

A vertex $x$ is said to be height-balanced if:

$$|x.\text{left.height} - x.\text{right.height}| \leq 1$$

An binary search tree is said to be height balanced if: every vertex in the tree is height-balanced.
Is this tree height-balanced according to AVL?

1. Yes
2. No
3. I am confused... 😞
Is this tree height-balanced according to AVL?

1. Yes
2. No
3. I am confused... 😞
Claim:

A height-balanced tree with \( n \) vertices has height \( h < 2 \cdot \log_2(n) \)

Proof (do not be scared):

Let \( n_h \) be the minimum number of vertices in a height-balanced tree of height \( h \)
Height-Balanced Trees

Proof:

Let $n_h$ be the minimum number of vertices in a height-balanced tree of height $h$

\[ n_h = 1 + n_{h-1} + n_{h-2} \]

$n_h > 1 + 2n_{h-2}$ (as $n_{h-1} > n_{h-2}$)

$n_h > 2n_{h-2}$ (obvious)

= $4n_{h-4}$ (recursive)

= $8n_{h-6}$

= $\ldots$

Base case: $n_0 = 1$
Height-Balanced Trees

Proof:

Let $n_h$ be the minimum number of vertices in a height-balanced tree of height $h$

\[
\begin{align*}
    n_h &= 1 + n_{h-1} + n_{h-2} \\
    n_h &> 1 + 2n_{h-2} \\
    n_h &> 2n_{h-2} \\
    &= 4n_{h-4} \\
    &= 8n_{h-6} \\
    &= \ldots
\end{align*}
\]

As each step we reduce $h$ by 2, then we need to do this step $h/2$ times to reduce $h$ (assume $h$ is even) to 0

Base case: $n_0 = 1$

\[
\begin{align*}
    n_h &> 2^{h/2} n_0 \\
    n_h &> 2^{h/2}
\end{align*}
\]
Claim:

A height-balanced tree is balanced, i.e. has height $h = O(\log(n))$

We have shown that: $n_h > 2^{h/2}$ and $n \geq n_h$

- $n \geq n_h > 2^{h/2}$
- $n > 2^{h/2}$
- $\log_2(n) > \log_2(2^{h/2})$ (log$_2$ on both side)
- $\log_2(n) > h/2$ (formula simplification)
- $2 \times \log_2(n) > h$ or $h < 2 \times \log_2(n)$

$h = O(\log(n))$
AVL Trees [Adelson-Velskii & Landis 1962]

Step 3: Show how to maintain height-balance
Insertion to an AVL Tree

```
insert(37)
```

Initially balanced

But no longer balanced after inserting 37

Need to rebalance!

But how?

“Infinite more” examples in VisuAlgo...
Rotations maintain ordering of keys

⇒ Maintains BST property (*see vertex B where* $P \leq B \leq Q$)

rotateRight requires a left child
rotateLeft requires a right child
BSTVertex rotateLeft(BSTVertex T) // pre-req: T.right != null

BSTVertex w = T.right
w.parent = T.parent
T.parent = w
T.right = w.left
if (w.left != null) w.left.parent = T
w.left = T

// Update the height of T and then w
return w

rotateRight is the mirrored version of this pseudocode

This slide is can be confusing without the animation
Balance Factor (bf(x))

\[ bf(x) = x.\text{left.height} - x.\text{right.height} \]

From the insertion point, check the balance factor of each vertex up to the root.

Once we have vertex with balance factor +2 or -2, we have to rebalance it.

-2, need rebalancing
Four Possible Cases

\[ \text{bf}(x) = +2 \text{ and } \text{bf}(x.\text{left}) = 1 \]
rightRotate(x)

\[ \text{bf}(x) = +2 \text{ and } \text{bf}(x.\text{left}) = -1 \]
leftRotate(x.\text{left})
rightRotate(x)

\[ \text{bf}(x) = -2 \text{ and } \text{bf}(x.\text{right}) = -1 \]
leftRotate(x)

\[ \text{bf}(x) = -2 \text{ and } \text{bf}(x.\text{right}) = 1 \]
rightRotate(x.\text{right})
leftRotate(x)
This is a case of -2, -1
Do left rotate on 29

“Infinite more” examples in VisuAlgo...
Rebalancing (2)

Now all vertices are balanced again

“Infinite more” examples in VisuAlgo AVL Tree Visualization
Summary:

- Just insert the key as in normal BST
- Walk up the AVL tree from the insertion point to root:
  - At every step, update height & check balance factor
  - If a certain vertex is out-of-balance (+2 or -2), use rotations to rebalance
    - During insertion to an AVL tree, you can only trigger one of the four possible rebalancing cases as shown earlier
Deletion is quite similar to Insertion:

- Just delete the key as in normal BST
- Walk up the AVL tree from the deletion point to root:
  - At every step, update height & check balance factor
  - If a certain vertex is out-of-balance (+2 or -2), use rotations to rebalance
  - The main difference compared to insertion into AVL tree is that you may trigger one of the four possible rebalancing cases several times, up to $h = \log n$ times :O, see this example (next slide)
Try Remove (Delete) vertex 7, it triggers two (more than one) rebalancing actions.

Then try various Insert operations and notice that at most it will only trigger one (out of the four cases) of rebalancing actions.
The Implementation

Let’s look *briefly* at AVLDemo.java

The code is **NOT** given, it is asked in PS2 😊

So, I will only flash them

Introducing **Java Inheritance and Polymorphism**

Q: Do we have to use such long code every time we need a balanced BST 😞?

A: Fortunately no ☺️, we can use Java API

Details during your lab demo on Week 04
Many different flavors of balanced search trees

- AVL trees (Adelson-Velsii & Landis, 1962)
  - Discussed in this lecture...
- B-trees / 2-3-4 trees (Bayer & McCreight, 1972)
- BB[α] trees (Nievergelt & Reingold 1973)
- Red-black trees (see CLRS 13)
- Splay trees (Sleator and Tarjan 1985)
- Treaps (Seidel and Aragon 1996)
- Skip Lists (Pugh 1989)
Balanced Search Trees

Red-Black Trees

- Every vertex is colored red or black
- All leaves are black
- A red vertex has only black children
- Every path from a vertex to any leaf contains the same number of black vertices.
- Rebalance using rotations on insert/delete
Balanced Search Trees

Skip Lists and Treaps

- Randomized data structures
- Random insertions \( \rightarrow \) balanced tree
- Use randomness on insertion to maintain balance
Balanced Search Trees

Splay Trees

- On access (search or insert), move vertex to root (via rotations)
- Height can be linear!
- On average, \(O(\log n)\) per operation (amortized)

Optimality?

- Cannot do better than \(O(\log n)\) worst-case
- What about for specific access patterns (e.g., 10 searches in a row for value \(x\))?
Now, after we learn **balanced BST**

<table>
<thead>
<tr>
<th>No</th>
<th>Operation</th>
<th>Unsorted Array</th>
<th>Sorted Array</th>
<th>bBST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Search(age)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>2</td>
<td>Insert(age)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>3</td>
<td>FindOldest()</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>4</td>
<td>ListSortedAges()</td>
<td>O(n log n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>5</td>
<td>NextOlder(age)</td>
<td>O(n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>6</td>
<td>Remove(age)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>7</td>
<td>GetMedian()</td>
<td>O(n log n)</td>
<td>O(1)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>8</td>
<td>NumYounger(age)</td>
<td>O(n log n)</td>
<td>O(log n)</td>
<td>????</td>
</tr>
</tbody>
</table>
Now, how to get $\text{rank}(v)$ efficiently?
This has not been discussed before and will be revealed during the live lecture.

$\text{NumYounger}(\text{age}) = \text{rank}(\text{age}) - 1$
Balanced BST

Summary:

- The Importance of Being Balanced
- Height Balanced Trees
- Tree Rotations
- AVL trees

Next Lecture:

- ADT Priority Queues
- Binary Heaps
CS2010R first meeting

• Tomorrow, Thu, 03 Sep 15, 6.00-6.30pm
• Note: venue TBA
• All R students must come
• Those who wants to get PS2E AC can also come 😊
• TA: Myself 😊